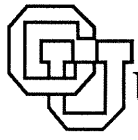
Scientific Programming Languages for Distributed Memory Multiprocessors

Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver

CU-CS-537-91 July 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUL 1991		**00-07-1991 to 00-07-1991**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Scientific Programming Languages for Distributed Memory Multiprocessors	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Colorado at Boulder,Department of Computer Science,Boulder,CO,80309-0430**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**37**	

Scientific Programming Languages for Distributed Memory Multiprocessors : Paradigms and Research Issues

CU-CS-537-91 July 1991

Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver

Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA

This research supported by AFOSR grant AFOSR-90-0109, NSF Grant No. CDA-8922510, and NSF Grant No. ASC-9015577.

Abstract

This paper attempts to identify some of the central concepts, issues, and challenges that are emerging in the development of imperative, data parallel programming languages for distributed memory multiprocessors. It first describes a common paradigm for such languages that appears to be emerging. The key elements of this paradigm are the specification of distributed data structures, the specification of a virtual parallel computer, and the use of some model of parallel computation and communication. The paper illustrates these concepts briefly with the DINO programming language. Then it discusses some key research issues associated with each element of the paradigm. The most interesting aspect is the model of parallel computation and communication, where there is a considerable diversity of approaches. The paper proposes a new categorization for these approaches, and discusses the relative advantages of disadvantages of the different models.

1. Introduction

This paper attempts to identify some of the central concepts, issues, and challenges that are emerging in the development of parallel programming languages for distributed memory multiprocessors. It describes a common paradigm for such languages that appears to be emerging, discusses some key differences between current approaches that highlight interesting research issues, and mentions some important needs that are only beginning to be, or are not yet being, addressed.

Our orientation is to languages that are being developed to utilize scalable distributed memory multiprocessors, such as hypercubes, for scientific computation. This orientation implies that the algorithms one wishes to express in the parallel programming languages are predominantly data parallel ones. By this we mean algorithms where at each stage, parallelism is achieved by dividing some data structure(s) into pieces, and performing similar or identical computations on each piece concurrently. The parallel language concepts and issues discussed in this paper are tied closely to this parallel algorithmic paradigm.

We also confine our attention to languages where the programmer takes some explicit actions to indicate the parallelism in the algorithm, as opposed to languages where the programmer specifies a purely sequential algorithm. In conjunction, we restrict our attention to imperative languages. Some examples of alternative approaches include Crystal [7,20], SISAL [22], and [27]. Finally, we are mainly oriented towards languages for MIMD distributed memory computation, but also refer to common issues in language support for SIMD distributed memory computers.

Recently, a number of parallel language research projects have been undertaken that share much or all of the orientation described in the previous two paragraphs. These include AL [36], Arf [37], C* [28], DINO [8,30], FortranD [9,15], Kali [18,23], Linda [6], Onyx [21], Pandore [2], Paragon [26], ParallelDistributedC [14,25], PC [3], Spot [35], Superb [12,38], Suspense [33]. Only some of these have so far led to completed and distributable languages and compilers.

In our view, the general goals of all these research projects are to make it easy to express data parallel algorithms for distributed memory multiprocessors, without causing significant degradation in the efficiency of the resultant code. They all seem to be motivated by three important points. First, the vendors selling such machines have generally provided only low level interprocess communication and process management primitives to support parallel computation, and these make the development of parallel programs far too difficult and error-prone. Second, the provision of compilers that automatically convert purely sequential programs into efficient parallel programs on distributed memory machines seems unattainable, at least in the near future. These two points make the higher-level, explicitly parallel languages that are the topic of all these projects a natural research target. The third point is that scientific users of these machines will probably only tolerate reasonably small degradations in efficiency, say 10-40%, if they are to use higher level approaches as opposed to the cumbersome low level primitives. This means that efficiency needs to be carefully considered from the outset in such research.

This remainder of this paper discusses some important general issues connected with this area of research. In Section 2 we briefly describe a common paradigm for distributed parallel languages that we believe is starting to emerge from these research projects. The key elements of this paradigm are mechanisms for specifying a virtual parallel machine and the distribution of data structures onto this machine, and the provision of some model of parallel computation and communication. Section 3 attempts to make these abstract concepts more concrete through one example, our own research on the DINO parallel programming language. This example also enables us to motivate many interesting research issues and challenges that we return to later. The remaining sections consider some key research issues and unanswered questions related to each element of the general paradigm. Sections 4 and 5 briefly consider distributed data structures and virtual parallel machines, respectively; on these topics the agreement between current approaches is fairly high. Section 6 discusses the model of parallel computation and communication, which appears to us to be the most interesting topic, for there are several significantly different approaches under consideration. These are described and

contrasted in some detail in this section. Some additional communication issues are briefly discussed in Section 7. Finally, in Section 8 we discuss an important research issue that is only barely being addressed by current research, namely high level language support for complex programs that may have multiple phases and/or multiple levels.

2. An Emerging Paradigm for Distributed Parallel Languages

Most current research in languages for distributed, data parallel computation appears to be aimed primarily at the level of a single algorithmic kernel. By this we mean a basic mathematical computation such as the multiplication or factorization of dense or sparse matrices, or a multi-color method for solving differential equations. At this level, the key features for support of parallel computation in most of these languages appear to be some forms of the following :

- Distributed Data Structures
- A Virtual Parallel Computer
- A Model of Parallel Computation with Associated Communication Model
- Additional Communication Features

The remainder of this section briefly defines and discusses each of these features, at an abstract level. The following section gives some concrete examples, in the context of the DINO language. Sections 4-7 then discuss research issues related to each of these language features.

Distributed data structures, meaning the ability to somehow indicate how data structures can be divided into parts that can be acted upon concurrently, are the key to data parallel computation. They are found not only in distributed parallel languages such explicit parallelism such as Kali [23] and DINO [30], but also in systems for annotating sequential programs to indicate potential parallelism to the compiler, such as FortranD [9]. The fundamental capability, found in almost all these

systems, is the ability to partition single and multiple dimensional arrays along one or more of their axes, with some languages allowing overlaps as well.

By a *virtual parallel computer* we mean some conceptual organization of processors, such as a single or multiple dimensional array, that may or may not correspond to the underlying target parallel machine. Almost any system that includes distributed data structures will require such a virtual computer, to serve as the target set of the mapping that defines the distributed data structure(s). The virtual parallel machine may be explicitly designated, or it may arise implicitly via the data distribution declarations. Languages that include explicit virtual parallel computers include DistributedParallelC [14] and DINO. The virtual parallel computer may also play a role in the model of parallel computation.

The *model of parallel computation* means the programming mechanism by which the programmer indicates operations that can be performed concurrently. Along with the associated model of communication, this appears to us to be the most interesting of the key constructs. This is because, while the capabilities of distributed data structures and virtual parallel machines in most proposed languages for distributed data parallel computation appear reasonably similar, a number of substantially different models of parallel computation and communication are being employed in these research projects. One key distinction between them is the view that is taken of the computation. One possibility is a "complete program view", where code similar to sequential code is written, describing the entire computation, and parallelism is designated via some annotations of the data declarations and possibly of the executable statements of the program. For example, the code may be purely sequential code with distributed data annotations, with the compiler attempting to infer parallelism from data dependency analysis combined with these annotations, or it may also use statements such as *forall* loops that indicate parallel execution. A second possibility is a "concurrent process view", where code is written that will be executed concurrently by each virtual processor. This may use a "per data element" view, where the programmer writes code that designates the operations to be performed upon each member of some distributed data structure(s), or a "per task" view, where the

programmer aggregates data element operations into the code to be performed by an actual or virtual processor.

Associated with the *model of parallel computation* is a model of communication, the method for designating the transfer of data between virtual processors. The three models that appear to be commonly used are SIMD (Single Instruction, Multiple Data), Block SIMD (also called "copy-in/copy-out"), and SPMD (Single Program, Multiple Data) with sends and receives explicitly designated in some manner. In the Block SIMD model, there are designated blocks of concurrent code, for instance the bodies of forall loops, with global synchronization implicitly enforced at the boundaries of each block, and all necessary communication of non-local values occurring at these boundaries.

The communication can either be explicitly designated or, more commonly, determined implicitly by the compiler from an analysis involving the code and the distributed data declarations. The SIMD model can be viewed as the block SIMD model where the blocksize is one operation. Again, communication usually is determined implicitly. In the SPMD send/receive model, the programmer does not designate global synchronization/communication points. Instead, the programmer generally may explicitly or implicitly designate communication between two or more processors at any point in the program.

In practice, these models of parallel computation and communication have been combined in most but not all ways. These included the complete program view with forall loops and block SIMD communication model, or the concurrent process view with with any of the three communication models. Clearly, each of these models has advantages and disadvantages in comparison to the others. These will be discussed in some detail in Section 6.

Additional communication features may include mechanisms that are important to parallel computation but are not captured naturally by these basic models. Common examples are reduction operators, operators for taking global sums, products, or extrema of distributed data structures.

Finally, we reiterate that this paradigm is most appropriate for a single algorithmic kernel, where in particular a single mapping of each data structure, and a single virtual computer, is likely to suffice. Complete computations may consist of sequences and/or nestings of such kernels, and thus may involve different distributions of given data structures, different virtual machines, and maybe even different models of parallelism at different stages. Additional language mechanisms may be required to accommodate this added complexity, but very few current research projects address this issue. Thus we consider this largely an issue for future research, and discuss it briefly in Section 8.

3. An Example of the Paradigm : The DINO Language

DINO (DIstributed Numerically Oriented language) is a language for distributed, data parallel computation developed at the University of Colorado [8,30]. It fits the paradigm presented in Section 2 and provides specific instances of each of its major features. In this section we use two sample DINO programs to illustrate these general features. We also use these examples to motivate some research issues concerning these general features. Where it is helpful, this discussion includes a critique of the relevant DINO constructs.

DINO consists of standard C, augmented by constructs for describing parallelism. The primary constructs are *structures of environments*, for describing virtual parallel computers; declarations of *distributed data structures* including facilities for describing the mapping of data structures onto virtual parallel computers; *composite procedures* for expressing SPMD parallel computations; and a combination of explicit and implicit means for specifying interprocessor communication.

Example 3.1 is a DINO program for multiplying an $N \times N$ matrix A by a N-vector x, using P processors. (It is complete except for the routines *init* and *display* for reading the data and displaying the results, respectively.) The algorithm it employs is to have each processor concurrently multiply a block of N/P contiguous rows of A by x, providing N/P elements of the result vector b. The statement "environment row [P: id]" declares a virtual parallel machine of P processors, named row_0,

$\cdots$, row_{P-1}, and the statement "environment host" declares a master processor named *host*. The three statements starting with the words "float distributed" declare the distributed data structures. They partition A and b into blocks of N/P contiguous rows and elements, respectively, mapping the i^{th} block onto the i^{th} virtual processor, and map all of x onto each virtual processor. The words *blockrow*, *block*, and *all* are predefined mappings provided by the language. The statement "composite matvec (in A, in x, out b)" declares a process called *matvec* that resides on each virtual processor row_i. When it is called by the statement "matvec (At[][], xt[], bt[])#" in *host*, the body of the composite procedure, i.e. the doubly nested for-loop, is executed on each virtual processor row_i using its portion of the distributed data. This provides an SPMD form of parallel computation.

The execution of this (and any) DINO program begins in the process *main* in the host processor. When *matvec* is called by *host*, the input parameters *At* are *xt* are distributed to the processors

---

Example 3.1 -- Matrix-Vector Multiplication in DINO

```
#define N 512
environment node [N:id]  {     /* defines array of N virtual processors  */
  composite MatVec (in M, in v, out a)     /*  Row-wise Parallel Algorithm to Calculate  a <-- M*v  : */
    float distributed M[N][N] map BlockRow;       /* maps row id of M to node[id] */
    float distributed v[N] map all;       /* maps v to each environment  */
    float distributed a[N] map Block;       /* maps a[id] to node[id] */
    {
    int j;
    a[id] = 0;
    for (j=0; j<N; j++)       /*  a[id]  <--  dot product of (row id of M) and v  */
      a[id] += M[id][j] * v[j];
  }  }
environment host   {   /* master process */
  main ()   {
    float Min[N][N];      /*  Matrix Multiplicand  */
    float vin[N];          /*  Vector Multiplicand  */
    float aout[N];          /*  Vector Answer  */

    InputData (Min);         /*  omitted for brevity  */
    MatVec (Min[][], vin[], aout[])# ;         /*  Call the Composite Procedure  */
    PrintResults (aout);         /*  omitted for brevity  */
  }  }
```

---

row_i according to the distributed data declarations of the formal parameters A and x, respectively. When all the processors row_i have concluded the execution of *matvec*, the output parameter b is collected into the variable *bt* of *host*. These are the only interprocessor communications required by this algorithm, and are accomplished without any explicit designation by the programmer.

Example 3.2 is an artificial smoothing program that is provided here to illustrate the one key feature of DINO that is not illustrated by Example 3.1, communication between concurrently executing processes in the SPMD model. This is accomplished in DINO using explicitly designated send and receive operations, which are specified by a reference to an element or subarray of a distributed data structure, followed by a # sign. In Example 3.2 the virtual parallel computer is the $N \times N$ array of processors *grid*. The distributed data structures are U and *maxderiv*. The mapping *FivePoint* maps element $U[i][j]$ to virtual processors $grid[i][j]$, $grid[i+1][j]$, $grid[i-1][j]$, $grid[i][j+1]$, and $grid[i][j-1]$. The composite procedure *smooth* defines the process that is executed by each virtual processor concurrently when it is called from *host*. The first if statement in *smooth* calculates a new value of each $U[xid][yid]$, by averaging it with the four neighboring values of U. By using the reference $U[xid][yid]\#$, it also sends this new value to each other processor to which $U[xid][yid]$ is mapped, namely $grid[xid+1][yid]$, $grid[xid-1][yid]$, $grid[xid][yid+1]$, and $grid[xid][yid-1]$. (The default paradigm used in DINO is that only the home processor, in this case $grid[xid][yid]$ for $U[xid][yid]$, sends out new values of this variable, while the others processors to which the distributed variable is mapped receive its new values.) The next two if statements calculate numerical approximations to the second derivative of U in the x and y directions, respectively. The references $U[xid+1][yid]\#$, $U[xid-1][yid]\#$, $U[xid][yid+1]\#$, and $U[xid][yid-1]\#$ on the right side of the assignment statements receive the new values of these variables that were sent out in the aforementioned send statement. These receives block until the new value of this variable is received from its home processor. This enforces local, produce-consume synchronization between the processors but no global synchronization. Finally, the operator *gmax* in the statement "maxderiv[] = gmax(myderiv[])#" is an instance of a reduction operator. This statement takes the maximum of each

myderiv [*i*] over all the virtual processors *grid* [*i*][*j*], and assigns this value to *maxderiv* [*i*].

Now we comment briefly on some interesting issues raised by these two examples. With regard to virtual parallel computers, note that in Example 3.1, the virtual parallel machine *row* has the dimension P that presumably would correspond to the underlying physical parallel machine, as opposed to the size N that would match the maximum parallelism of the algorithm. In contrast, in Example 3.2 the dimensions of the virtual parallel machine *grid* match the full data parallelism of the algorithm, in this case $N \times N$. We have purposely used these two different styles in the two examples, to raise the general issue of whether the programmer needs to cater the degree of parallelism of the

---

Example 3.2 -- Two Dimensional Smoothing Illustrating DINO Communication Features

```
#define N 128
environment grid [N:xid] [N:yid]  {     /*  defines N x N grid of virtual processors  */
    composite Smooth (in U, out maxderiv)       /* parallel process executed on each environment */
        float distributed U[N][N] map FivePoint;       /* maps U[x][y] to grid[x][y] and four neighbors */
        float distributed deriv_max[2] map all;       /* mapped to all environments */
        {
        float myderiv[2];
        if ((xid != 0) && (yid != 0) && (xid != N-1) && (yid != N-1))     /* if not a border point */
            U[xid][yid]# = (U[xid][yid] + U[xid-1][yid] + U[xid+1][yid] +  U[xid][yid-1] + U[xid][yid+1]) / 5;
            else U[xid][yid]# = BorderSmooth (U, xid, yid);     /* omitted for brevity */
        if ((xid != 0) && (xid != N-1))     /* approximate second derivative in x direction */
            myderiv[1] = U[xid][yid] - (U[xid-1][yid]# + U[xid+1][yid]#)/2 ;
            else myderiv[1] = 0;
        if ((yid != 0) && (yid != N-1))     /* approximate second derivative in y direction */
            myderiv[2] = U[xid][yid] - (U[xid][yid-1]# + U[xid][yid+1]#)/2 ;
            else myderiv[2] = 0;
        maxderiv[] = gmax(myderiv[])#;      /* reduction operator on myderiv[1] and myderiv[2] */;
    } }
environment host   {   /* master process */
    main ()   {
    float U [N][N];
    float maxderiv [2];

        InputData (U);      /* omitted for brevity */
        smooth (U[][], maxderiv[])# ;        /*  Call the Composite Procedure  */
        printf(" Maximum Second Derivative of Smoothed Data in x Direction is %.3f", maxderiv[1]);
        printf(" Maximum Second Derivative of Smoothed Data in y Direction is %.3f", maxderiv[2]);
    } }
```

---

parallel program to the parallel computer, or just to the parallel algorithm. In DINO one can do either but programs that have more virtual than actual processors usually will be inefficient. In general this issue is closely tied to the model of parallel computation that is used, and thus will be discussed in Sections 4 and 6.

One noteworthy feature of distributed data structures in DINO is that they are referred to using global rather than local names. A second is that the mappings of these data structures onto virtual parallel machines allow replication, i.e. the mapping of a particular data element to multiple processors, as illustrated by Example 3.2. This has possible implications for interprocessor communication and storage management. A third issue is the variety of mappings that are provided; DINO provides a fairly rich general mechanism for describing mappings that is not illustrated by these examples [ref]. These are among the issues that will be discussed in Section 5.

Parallelism execution in DINO arises from calls to composite procedures, which provide a concurrent process model of computation. The programmer writes code for one process, using either a per task or per data element view of the computation in the terminology of Section 2. An alternative approaches that we are currently pursuing is discussed in Section 6.

The examples illustrate that there are two main interprocessor communication mechanisms in DINO. The first is that communication arises implicitly when distributed data structures are used as input, output, or input/output parameters to composite procedures. This communication is implicit and follows a block SIMD model, with the data structures being distributed or collected automatically at the composite procedure invocation or termination, respectively. The second mechanism is the explicit designation of communication between processes through the use of # signs following references to distributed variables. The new value of the distributed variable is either sent to all the processors it is mapped to (as defined by the distributed data declaration), or received from its "home" processor, using a produce-consume paradigm. While the syntax of these statements is much simpler than the low level interprocessor communication statements provided by vendors, they still require the programmer to explicitly specify communication. A key issue that will be addressed in Section 6

will be the connection between the model of parallelism used, the need for explicit specification of communication, and the expressiveness and parallel efficiency of the language.

Finally, the reduction statement (*gmax*) in Example 3.2 illustrates two additional communication features in DINO. One is reduction operators, and the second is the addition of array operations that allow communication primitives to be applied to arrays or sub-arrays. These and other specific communication issues are addressed in Section 7.

Many features of DINO are not illustrated in this section. Some important ones include the ability to specify multiple virtual parallel machines, to support non-blocking as well as blocking receives, and to support functional parallelism. Complete descriptions are available in [8,30,32]. The DINO compiler is freely available from the authors.

4. Research Issues Regarding Virtual Parallel Computers

It appears likely that most languages for distributed, data parallel computation will include an explicit or implicit designation of virtual parallel computers. This seems to be an integral part of the definition of distributed data structures, which is at the heart of these languages, and may be used in the model of parallel execution as well. This section discusses some important issues regarding language constructs for virtual parallel computers that have been raised by current research, and then mentions some future opportunities regarding this language construct.

A key issue illustrated by the examples in Section 3 is whether the virtual parallel computer should correspond to the underlying target parallel machine, or only to the parallel algorithm. We have purposely used Example 3.1 to illustrate the former approach, which uses a P processor virtual computer and a "per task" algorithm, and Example 3.2 to illustrate the latter approach, using an $N \times N$ processor virtual machine and a "per data element" algorithm. In DINO, either approach is possible, but currently an algorithm that has more virtual than actual processors will be implemented by

multiprogramming the processors, which is likely to be inefficient.

In our view, it is highly desirable to allow the virtual parallel machine to correspond to the parallel algorithm and be independent of the target parallel machine. In order for this approach to result in efficient code, however, it almost certainly is necessary to be able to *contract* the many virtual processes. By this we mean that the compiler must efficiently be able to produce an equivalent, efficient parallel program whose number of processes equals the number of processors of the target machine. Whether this is possible is strongly dependent on the communication model that is used in the language. In languages using SIMD and block SIMD models, which provide a global view of all synchronization and communication, contraction is relatively straightforward. In languages where synchronization and communication can be designated between arbitrary pairs of processors at arbitrary points, such as DINO or any language employing a full SPMD send/receive model, automatic contraction seems more difficult and sometimes may be infeasible. On the other hand, there are limitations to the expressiveness of languages that only support SIMD or block SIMD communication. This general issue is discussed in detail in Section 6.

A more minor, related issue is whether the number of processors in the target parallel machine needs to be known at compile time or only at run time. For example, in Example 4.1 the question becomes whether the program needs to be recompiled each time the define statement for P changes. (In current DINO, the answer is yes.) In a fully data parallel program, the program is independent of the target machine, but the question becomes whether the compiler needs to know the target number of processors to produce an efficient, *contracted* program. From our experience in writing compilers for parallel machines, we suspect that it will be very difficult to produce efficient parallel programs from high level language descriptions without recompiling for each different number of processors. To our knowledge, all existing compilers do require the number of processors to be known at compile time.

A final issue regarding the facilities of current languages for virtual parallel machines is the mapping of the processors of the virtual machine to the processors of the target parallel computer. In

almost all current languages, including DINO, the virtual machines must be single or multiple dimensional arrays of processors. In this case, the obvious strategy is to maps contiguous virtual processors (whose indices differ by one in exactly one index) onto contiguous actual processors. This is fairly straightforward to accomplish, see e.g. [16]. As long as most communications occur between pairs of contiguous virtual processors, as is generally the case, this simple strategy also is efficient. If communication patterns are more random, however, then determining an efficient mapping of virtual to actual processors may be very difficult and may even require programmer guidance. Implicit in this discussion is that it is preferable for communication to occur between contiguous actual processors. The importance of this has varied among recent distributed architectures, but we assume that in a congested situation where many pairs of processors are communicating simultaneously, almost any architecture will perform more efficiently when utilizing contiguous rather than noncontiguous communications.

An important challenge that we see regarding language constructs for virtual parallel computers is the provision of more general virtual parallel machines than single and multiple dimensional arrays. This could include virtual parallel machines that are defined statically but use more general data structures, for example fixed undirected graphs, as well as virtual parallel computers that are defined dynamically as the computation proceeded, such as dynamically generated trees. While such features are not required for many scientific computations, such as those based on dense, regular grids or dense matrices, they may be useful for less regular computations, such as adaptive grid methods or particle physics algorithms. Key questions regarding such constructs include : What language constructs are most natural for expressing such virtual parallel computers? How do such constructs, especially dynamic virtual machines, interact with distributed data declarations and models of parallel execution? How are such virtual machines mapped to the actual parallel computers? How efficient will the resultant code be? These are all difficult questions. In particular, whether it will be possible to produce *efficient* parallel programs from descriptions that use dynamic virtual parallel machines is an open and challenging research issue.

5. Research Issues Regarding Distributed Data Structures

As discussed earlier, the provision and use of distributed data structures appears to be the key and fundamental element of parallel programming languages for distributed, data parallel computation. So far, most distributed language research projects have taken reasonably similar approaches to distributed data structures. These are predominantly based on making fixed mappings of regular arrays of data onto regular arrays of virtual processors. In this section we briefly discuss issues relating to this current research, and then discuss future challenges. These are mainly related to accommodating less regular data structures, and allowing mappings to change as the algorithm proceeds.

Many languages support, in some form, the mappings of single or multiple dimensional arrays of data onto virtual parallel machines that are also single or multiple dimensional arrays. Generally they allow each axis of the data structure to be partitioned onto a given axis of the virtual parallel machine, and in most cases, support *block*, *wrap*, and sometimes *block-wrap* partitionings. Most systems also allow certain data structures to be replicated among all the virtual processors.

One distinction between current languages is whether they allow "overlaps", i.e. the mapping of elements of the data structure onto multiple virtual processors, or just partitions of the data structures. Languages that support overlap include Superb [38], FortranD [9], and DINO (see Example 3.2). Where overlap mappings are provided, they are generally used to provide information concerning communication and storage to the compiler. For example, they may tell the compiler that processors $i-1$, i, and $i+1$ all store the value of x_i, and that when a new value of x_i is generated it must be sent to each of these processors. This type of sharing of data is fairly common in scientific computation. Languages that do not use overlap mappings rely instead on the compiler to determine the communication and storage requirements from the code. It will be interesting to see which approach is preferred by users and by language and compiler developers.

An interesting distinction between current approaches is whether they use a global or local name space to refer to elements of distributed data structures. For example, if a program partitions a

data structure y with n elements onto a array of n processors, does the program for processor i refer to its element as y_i, or y_0, or just plain y? Most languages have used the former, global, approach, and we find this to be more natural in most cases, for it usually causes the program to correspond more closely to the sequential code that the programmer is used to.

In our opinion, two key future challenges concerning distributed data structures are remapping distributed data structures as the computation proceeds, and handling irregular data structures. Both have only had preliminary consideration in work that has been done so far.

In the context of making regular mappings of arrays data structures onto arrays of virtual processors, it is fairly straightforward to provide language features for remapping the data by somehow changing the distributed data declaration. Several recent proposals [9,29,31] incorporate this feature in some form. It will be interesting to see how efficiently these remappings can be implemented and how general they can be in practice. In the context of irregular data structures, remapping is just part of the larger challenge that we discuss next.

Mapping and accessing irregular distributed data structures is a difficult but important challenge. It is important because irregular data structures are necessary to efficiently model many scientific phenomena, such as the behavior of shock waves in fluids. The approach that is taken to incorporating irregular data structures into parallel languages is likely to be related to the underlying language. In Fortran, irregular data structures are represented by indirectly indexed arrays, and some recent research projects are considering incorporating these data structures into languages for distributed memory multiprocessors [kali, arf, fortrand]. Some of the key issues that need to be addressed in this context are how the array is mapped to the virtual parallel machine, and how efficiently indirect accesses to the array can be resolved, including generating communications as required that we will mention briefly in Section 6. A series of papers from ICASE [17,24,34] has begun to address many of these issues.

In many other languages, irregular data structures are represented by pointer-based data structures. To our knowledge, work aimed at using pointer-based structures as distributed data structures in languages for distributed memory MIMD computation is just beginning. An interesting distinction between the two approaches is that pointer-based structures naturally impose a local view of the data, whereas indirectly accessed arrays utilize an underlying global name space. This is likely to have implications upon the programming model. Another interesting issue in using distributed pointer-based data structures is whether the associated virtual parallel machine is an array, or an undirected graph tied to the data structure. If the virtual parallel machine is an array, then a key question appears to be how one maps the pointer-based data structure onto this virtual machine. If the virtual parallel machine corresponds to the data structure, then the problem is the mapping of the virtual machine onto the physical machine in a way that gives good load balancing. The latter approach may be necessary if one is to naturally express parallel algorithms involving complex data structures that change dynamically as the algorithm proceeds. Some of our current research is addressing these issues in the context of the DINO language.

6. Research Issues Regarding Models of Parallel Computation

Inherent in any parallel language is a model of parallel computation and an associated model of communication. By "model of parallel computation" we mean the view that the programmer takes of the concurrent execution of the code. By "model of communication" we mean the view that the programmer takes of the transfer of data between virtual processors. Whereas most languages for data parallel scientific computation so far have taken similar approaches to the other key aspects of the parallel language, distributed data structures and virtual parallel machines, they have taken several rather different approaches to models of parallel computation. This section describes and contrasts these approaches. Then it very briefly mentions a new approach that we are taking in our research.

The main models of parallel computation that are being used in imperative languages for data parallel computation on distributed memory multiprocessors appear to be :

Annotated Complete Program -- The programmer writes standard serial code with one of the following two levels of annotation. In both cases, the programmer writes the instructions of the entire algorithm.

Data Distribution Annotations -- The programmer gives indications to the compiler of how data structures should be partitioned among actual or virtual processors. The compiler then determines how to parallelize the program.

Data Distribution and Parallel Execution Statements -- In addition, the programmer indicates how the program can be parallelized, generally by indicating loops that can be performed in parallel. The compiler may or may not attempt to find additional parallelism beyond that indicated by the programmer.

Concurrent Process -- The programmer writes code that will be executed in parallel by each (virtual) processor. Thus the programmer only writes the instructions for a typical portion of the algorithm. (The indices of the virtual processor may be used in the code.) There appear to be two main ways in which this may be done :

Per Data Element -- The programmer writes code at the full level of data parallelism for the particular algorithm. That is, the number of virtual processes equals the amount of data parallelism.

Per Task -- The programmer writes the code that aggregates a number of data parallel operations into a task. This occurs most commonly in models where the programmer writes the code for one actual processor.

If the programming model is complete program with only distributed data annotations, then the programmer is not concerned with any communication model. In all other cases, some communication model must be used along with the model of parallel computation. Three communication models appear to be in common use in imperative languages for data parallel computation on distributed

memory multiprocessors :

SIMD (Single Instruction Multiple Data) -- The communications are the same as if the code were executed in lock step using standard SIMD semantics.

Block SIMD -- For some designated block of parallel code, all off-virtual-processor values required within the block are obtained immediately preceding the beginning of the block, and all off-processor values generated within the block are communicated immediately following the end of the block. (This model is sometimes called "Copy-In/Copy-Out".) If the size of the block is a single operator, then this is the SIMD model.

SPMD (Single Program Multiple Data) with Sends/Receives -- Sends and receives between virtual processors may be designated at any point in the code. Typically they obey produce/consume semantics.

The SIMD and Block-SIMD models can use either explicit (programmer designated) or implicit (compiler determined) communication. Typically, they use implicit communication because it appears to take some of the work away from the programmer. As we discuss below, the SPMD send/receive model must use explicit communication.

In theory the choice of a communication model is orthogonal to the choice of a computation model. In this section, we only discuss those combinations we have seen. They are the annotated complete program computation model with block SIMD communications, and the concurrent process computation model with any of the three communication models.

We note that the above categorization of models of parallel computation and communication is new, and that we have selected the names for the categories. Quite possibly, these names could be improved. The terms "annotated complete program" and "concurrent process" are new in this context and alternatives clearly exist. The acronyms SIMD and SPMD are well established but generally connote models of computations as well as communication. Here we are using just the communication part of their meaning. An alternative set of terms of the three communication models could be

"operation synchronous", "block synchronous", and "asynchronous produce/consume".

THE ANNOTATED COMPLETE PROGRAM MODEL

The idea of deriving a parallel program for distributed memory machines from a serial code annotated solely by distributed data declarations was discussed by Callahan and Kennedy [5] and is the basis of the Superb [38] and Pandore [2] languages. It also is at the heart of the recent FortranD proposal [9] although this proposal also allows for parallel execution annotations in the form of forall loops. Presumably, serial programs annotated solely with data distribution declarations are easier to write than explicitly parallel programs, and clearly any algorithm can be expressed in this paradigm. The view the programmer takes is of the entire problem, as in any sequential language. The compiler has the entire job of detecting potential sources of parallelism, and the associated required communications, by using data dependency analysis, and of using this information to generate a parallel program that is equivalent to the serial one. The data distribution annotations may help the compiler to identify parallelism, and also aid it in distributing the problem to the processors once parallelism is identified. The key question regarding this model is how well a sophisticated compiler will be able to identify parallelism, and generate efficient communication, and how efficient the resultant parallel program will be in comparison to explicitly parallel programs for the same algorithms. Currently this is an open research issue.

In the other variant of the annotated complete program model, the programmer also provides parallel execution annotations. Typically this means that particular loops are flagged for parallel execution, for example by using a "forall" or "*DO" statement in place of a standard "do" or "for" loop. The implication of this annotation is that the compiler may assume that there are no inter-iteration dependencies, and thus that it may execute all the iterations of the loop in parallel. If there are inter-iteration dependencies, then the result of the parallel execution may be different than if the code were executed serially. Thus the programmer must understand the implication of the parallel loop annotation. Languages that use this model include AL [36], Arf [37], Kali[23], and FortranD [9]. In

contrast to FortranD, most of these do not attempt to have the compiler detect additional parallelism beyond that specified by the programmer.

Since the semantics of parallel loops in the annotated complete program model assumes that there are no inter-iteration dependencies, the natural communication model is the Block SIMD model, where the block is the body of the parallel loop. Many of the languages mentioned above use this communication model. It is the compiler's job to assure that the block SIMD semantics are followed, meaning that if a virtual processor refers to a variable residing on another virtual processor, the value that is obtained is the one that variable had just before the start of the parallel loop. Similarly, if any virtual processor assigns a new value to a variable residing on another virtual processor, the compiler must send that value to that variable immediately after the conclusion of the forall loop. Implementing these rules in a way that efficiently aggregates communication and minimizes processor idle time awaiting communications is a challenging research issue, but one that has already been extensively investigated by some researchers (see e.g. [17,21,24,34]). This research has included the difficult case of irregular data structures that are specified through the use of indirect index arrays in Fortran.

While there is limited experience in using the annotated complete program model, it appears to be fairly easy to use for many regular applications. No communication is specified by the programmer, and the Block SIMD semantics make it fairly easy for the compiler to determine where communications are required. The model also fits the loosely synchronous model of parallel computation, which has been said to apply to a good portion of parallel scientific algorithms [10, pp. 43-44]. On the other hand, the Block SIMD semantics may be confusing to the programmer, and may force the programmer to break parallel loops in unnatural places to force communications. An example of this is given below. Finally, there are certainly problems to which the block SIMD communications model does not apply, and for which it is difficult or impossible to obtain an efficient parallel algorithm from an annotated serial program.

THE CONCURRENT PROCESS MODEL

The concurrent process computation model has been used by a number of research projects into data parallel languages for distributed memory machines, in conjunction with all three communication models. In each case, the programmer writes code for one typical virtual process that will be executed concurrently by each virtual processor.

An example of a language that uses a concurrent process computation model together with a SIMD communication model is DataParallelC [14]. The SIMD communication model has its origins in languages developed for SIMD machines, such as C* [28] for the Connection Machine. In SIMD languages, one generally writes code for one generic virtual process that is then executed concurrently, in lock step, by each virtual process. Thus the programmer's view of the problem is per virtual process, which is often the same as per data element. Typically, no communication is specified by the programmer. Due to the lock step semantics, it is fairly easy for the compiler to analyze where communications must occur. The experience in using languages such as C* seems to be that such programs are easy to write.

When SIMD languages are used on MIMD machines, they do not need to synchronize at each operation, rather just at the communication points that are inferred from the SIMD semantics. Thus they can lead to efficiently executing programs even on machines where communication is expensive relative to computation. In order to do this, there must be enough computation available to hide the latency of the required inter-processor communications. Various optimizations also may be used to reduce the communication overhead. One optimization is to utilize the high degree of virtual parallelism to perform the computation for those virtual processors that do not require any off-processor communications while waiting for the required communications. Another optimization is to use extensive dependency analysis to to eliminate unnecessary communication points, pre-send data, and aggregate sends.

The big disadvantage of the SIMD form of the concurrent process model is its limited expressiveness. Only a subset of data parallel algorithms are naturally expressed as SIMD algorithms. Some other algorithms may be coerced to fit this model by specifying that certain processors do nothing at certain steps ("masking"), but this reduces the efficiency of the parallel algorithm in a way that may be totally unnecessary for an MIMD machine. Examples of algorithms which do and don't fit the SIMD model are given below.

As discussed in Section 3, the DINO language uses a concurrent process model with both block SIMD and send/receive forms of communication. The communications that results from using distributed data as parameters to composite procedures follow a block SIMD communication model, where the block is the composite procedure. The communication semantics are the same as those discussed above for forall loops. The main distinction is that in DINO, the programmer specifies which data may need to be communicated, by including them as parameters. Thus the compiler needs to do less work to determine communications than for forall loops, but the possible communication patterns are restricted to distributed data mappings whereas in forall loops there is no restriction. Another language that uses the concurrent process model with (only) block SIMD communication is Spot [35]. In this language an iteration is a basic programming construct and blocks correspond to single iterations.

Communications between concurrently executing processes in DINO uses the SPMD send/receive communications model. This is also the model used in low-level programming systems supplied by vendors of distributed memory MIMD machines, and in a variety of portable systems that have been produced including PICL [11] and PVM [4]. In the concurrent process model with send/receive communications, the programmer writes code that will be executed by each virtual or actual processor on its part of the data structure concurrently. In contrast to the SIMD model above, the instructions of each process are not assumed to execute in lock step. Rather, communications generally are specified explicitly, such as with the # operator in DINO. Synchronizations between processors, if any, are determined by the requirements imposed by these communications. Because the

communication patterns may be completely arbitrary (as opposed to the SIMD or Block SIMD communications models), it appears infeasible to have the programmer omit the specification of communications and have the compiler determine where they are needed. For related reasons, it may be difficult to efficiently contract programs written in this model if the parallelism in the program is greater than the actual parallelism.

Because the programmer needs to explicitly specify communications, the concurrent process model with general send/receive communications may be more difficult to program in than the other models. On the other hand, this model is more expressive than the other parallel models discussed (except serial programs with just data annotations) and may be more likely to lead to efficient parallel code for difficult or irregular parallel algorithms. An example of this is given later in this section.

DISCUSSION

The annotated complete program model, especially with only data distribution annotations, is particularly attractive because it lends itself to reuse of existing code and because a programmer used to an existing serial language has fewer new things to learn to do parallel programming. It should be realized, however, that the programmer will sometimes still have to understand the parallel aspects of the program if it is to run efficiently. Not all algorithms parallelize easily. The programmer may have to understand what the compiler is doing to a particular annotated serial program to be able to determine which annotations will allow the compiler to produce an efficient parallel program. In addition, it is still an open research question how well a parallelizing compiler can be made to perform on distributed memory machines using just information provided by data distribution annotations.

The concurrent process model has exactly the opposite problems. It does not lend itself as well to reuse of serial code, and it requires the programmer to learn more new concepts. Additionally, many current languages using this model may require the programmer to specify things that a compiler could figure out. The long term result may be research focusing on ways to combine these two models.

An example of something that the compiler often can determine is communications. It is a widely held opinion, and probably the correct one, that all other things being equal it is better to relieve the programmer of the chore of designating communications. The gain from doing this, however, may not as large as one might think at first glance. The problem is that to understand and improve the performance of any modestly complex program, the programmer still has to understand where communications and synchronizations are occurring. Thus the programmer may only be relieved of the chore of designating the communications, not of understanding where they occur. To complicate this trade-off, with the current state of the art in compilers, having the compiler determine the communications may result in loss of efficiency.

A more subtle difference between the annotated complete program model and the concurrent process model is the relationship between the mapping of virtual processes and distributed data to the virtual parallel machine. In the concurrent process model, both processes and data are mapped together to the same virtual machine. In the annotated complete program model, data is mapped to a virtual machine, and concurrent processes may be specified in forall loops, but they are not mapped to the same virtual machine. Instead the compiler has the job of aligning the data and processes. This means that the programmer has to specify less, and has less control. Experience will show which approach is advantageous.

The discussions of these models illustrates what we believe is the key dilemma in choosing the model of parallel computation and communication to use in a parallel programming language : the tradeoffs between ease of use (and re-use), expressiveness, and parallel efficiency. Ideally, one would like the model to be easy to use while still leading to efficiently executing parallel programs for a wide variety of data parallel algorithms. Unfortunately, among the parallel models we have discussed, there appears to be an inverse relationship between ease of use and the breadth of efficient parallel algorithms that they can naturally express. The next section illustrates the expressiveness of the various models with three examples. Then we will conclude this section with a brief discussion of one possible, hybrid, approach to combining expressiveness, ease of use, and parallel efficiency in a

concurrent process model.

EXAMPLES

As the first example, consider an LU decomposition algorithm for solving dense systems of linear equations. Assume that the matrix is distributed among the (virtual) processors by columns, which is the arrangement most conducive to parallel efficiency. Then the desired parallel algorithm is:

For j = 1 to n-1

Processor with column j determines pivot row and multipliers and broadcasts this information

All processors pivot their columns and perform elimination on columns > j

A language using the concurrent process model with SIMD communications can express this algorithm easily, using a mask for the pivot step. A language using the concurrent process model with send/receive communications also has no problem expressing the algorithm, but the programmer must explicitly designate the communications which seems unnecessary for this algorithm. A compiler for a serial language using data distribution annotations will almost certainly produce an efficient parallel algorithm for this example. If the annotated complete program model with forall loops and block SIMD communications is used, however, an interesting and subtle issue arises. The programmer might place the forall loop around the entire iteration (pivot/multiplier plus elimination), with a statement like "if mycolumn = j then do pivot/multiplier calculation" for the first segment of the iteration. This program would not work correctly, however, since communication would be required in the middle of the iteration and this would not occur using the block SIMD model. Instead, one would need to write the pivot/multiplier segment as a sequential block of code and place just the elimination segment in a forall loop. While there are more convincing examples, this illustrates the need to accommodate the block SIMD communications semantics when placing forall loops in annotated complete program model.

As a second example, consider a simplified view of an algorithm for solving a system of n nonlinear equations in n unknowns, using finite difference Jacobian matrices. There are two basic and potentially expensive steps at each iteration : calculating the finite difference Jacobian matrix by performing n evaluations of the user-provided system of nonlinear equations (which are independent of one another and thus can be performed concurrently), and performing an LU decomposition of this matrix as discussed in the first example. A serial program annotated only with data distributions is unlikely to be able to successfully parallelize the finite difference Jacobian evaluation in many applications, since each function evaluation may be a call to a subroutine that contains thousands of lines of code, and it may be impossible to determine that these function evaluations are independent. Adding a forall annotation to the loop for these n function evaluations would, however, permit them to be performed in parallel. It may still be difficult for the compiler to determine what communications is required when in fact, very little is. A concurrent process model with SIMD communications will not be able to express this algorithm unless the nonlinear function can be executed concurrently in lock-step, which is rarely the case in practice. (For example the function is commonly a differential equations solver that uses variable time steps.) A concurrent process model with send/receive communications will have no problem expressing the finite difference Jacobian evaluation, although again it will require some explicit communication designations that other models do not. We note that the new model we discuss at the end of this section is very well suited to this example.

As a final example, consider a distributed triangular solve (forward or back solve) using a matrix that is distributed by columns. This situation arises commonly in practice, for example in parallel algorithms for solving systems of nonlinear equations. The standard sequential triangular solve algorithm does not parallelize effectively, but researchers have constructed pipelined algorithms that are equivalent to the standard algorithm on serial processors but obtain more parallelism on distributed memory machines [19]. These algorithms are not described in any natural way by annotating a complete program with forall loops, or by using the concurrent process model with SIMD communication. They certainly would not be derived by any compiler from a serial algorithm with just data

distribution annotations. It appears that, among the models we have considered, only a concurrent process model with explicit sends and receives permits a reasonable representation of this algorithm.

Hopefully, these examples illustrate that no parallel language model is optimal with respect to both ease of use and expressiveness. They also indicate that in any model, the programmer may often have to think about parallelism and communication. Finally, it is reasonable to expect that many complex numerical algorithms will have aspects of all three of these examples. This makes it clear that providing a language that easily leads to efficient parallel programs for complex applications is a very challenging problem.

A NEW HYBRID COMMUNICATIONS MODEL FOR EXPLICITLY PARALLEL LANGUAGES

As part of our current research, we are attempting to develop a language incorporating the concurrent process model of parallel computation that combines ease of use, expressiveness, and parallel efficiency to a greater extent than those described above. Its basic new component is a "pseudo-SIMD" communications model. This model differs from the standard SIMD model in that it allows the programmer to include arbitrary, standard function calls as statements, with the semantics that these calls are executed concurrently and independently, that is without SIMD semantics. Communications may be required to obtain the values parameters to the function before it is called, but otherwise each virtual process performs its function call independently. With this small extension to the SIMD model, many data parallel algorithms can be written easily and with no explicit designation of communication. For instance, the first two examples above are handled easily and naturally using this model. To be very broadly expressive, however, we feel it is also necessary to provide the possibility of explicitly designated sends and receives. In our current research we allow each concurrent process segment (composite procedure in DINO) to use either pseudo-SIMD or SPMD semantics, but not both, and we allow the parallel program to be composed of arbitrary combinations of these two types of composite procedures. This allows the send/receive model to be used only where necessary and in an encapsulated manner. This new model is described in [29]. It will be interesting to see

whether future research can come up with a cleaner model that provides for ease of use, expressiveness, and parallel efficiency.

7. Additional Research Issues Regarding Communication Features

In all of the language approaches discussed in this paper, the programmer has to think to some extent about communication. The most important ways this is done is via the models of communication discussed in Section 6, and when making the distributed data declarations discussed in Section 5. There are some additional issues of general interest in designing languages, compilers, and run-time support systems for data parallel languages for distributed memory multiprocessors. This section very briefly mentions some of these.

Several communication features that are useful in data parallel computation are supported by the hardware or low-level software of most distributed memory machines. These include reduction operators (global sums, products, maximums, and minimums), and non-blocking receives such as are used in "chaotic" parallel algorithms. An interesting issue is whether, and how, parallel languages provide access to these features. Reduction operators appear crucial to data parallel computation. They are easy to provide in any language that uses an concurrent process model, and almost all do so. It may prove useful to provide them explicitly in annotated serial languages as well, and FortranD, for example, does so. It is not as clear how important it is to provide access to non-blocking receives, and this probably only is easy in languages with some form of explicit sends and receives.

There are several issues that a compiler and run-time system for a production quality distributed data parallel language needs to consider. These are all influenced by the fact that communication latency on distributed memory machines is relatively high, and can reasonably be expected to remain so. One issue is pre-sending communications, in order to eliminate the potential idle time that can result if a processor must wait for a message. In any language that does not explicitly designate sends, pre-sending communications requires analysis by either the compiler or the run-time system.

A second issue is aggregating communications to reduce the total communications latency. In the annotated complete program model or the concurrent process model with SIMD or block SIMD communications, most aggregations appear to require similar analysis to that used to vectorize arithmetic computations. Thus this process should be quite feasible. A final, more subtle issue is that many distributed memory machines turn out to have various low-level communication methods that have different latency costs, for reasons such as differing usages of low-level buffering, or different utilizations of special hardware features. Language and compiler designers will ultimately need to consider how they can cause their communications to utilize the most efficient low-level communications methods in most cases. They will also need to interact with multiprocessor vendors to encourage them to provide hardware and low-level software features that are consistent with the needs of parallel languages.

8. Research Issues Regarding Support for Complex Parallel Programs

The preceding discussion, and almost all research so far in language support for distributed, data parallel computation, is primarily oriented at supporting single algorithmic kernels, such as a matrix factorization algorithm or a conjugate gradient solver. As even the simple examples in Section 6 illustrate, real applications generally are composed of many such kernel algorithms. These algorithms may follow one another serially in the computation, or may be nested within one another, or both. A number of additional issues arise when considering language support for such complex computations. This section very briefly mentions some of these. Among the research projects currently considering these issues are the Orca project at the University of Washington [1,13] and the DINO project [29, 31].

A complex parallel algorithm may have serially executed phases, concurrently executed phases, and/or nested phases. Different phases may be expressed naturally using different virtual parallel machines. Thus one issue that arises immediately is whether and how to support serially changing,

multiple, or nested virtual parallel machines. The latter two cases raise challenging problems with regard to contraction and load balancing.

A related issue is that different phases of a complex algorithm may most naturally desire different mappings of the same data structure. One question this raises is how the language will support remapping of distributed data structures. This question is addressed in the recent FortranD proposal [9,15] and to a more limited extent in [29]. A second question is how efficiently the compiler will be able to perform remappings. This may require a library of common remapping operations that are optimized with regard to communication costs.

Finally, almost all research so far into distributed data parallel languages has been oriented towards algorithms that use static process structures and, for the most part, static data structures. Dynamic parallel algorithms, such as those used in adaptive grid methods for partial differential equations, pose many additional challenges that have been discussed somewhat in Sections 4 and 5. It remains to be seen whether these will be effectively addressed using the frameworks discussed in this paper.

9. References

[1] Alverson, G.A., Griswold, W.G., Notkin, D., and Snyder, L. A flexible communication abstraction for nonshared memory parallel computing. *Proc. of Supercomputing 90,* 1990, 584-593.

[2] Andre, F., Pazat, J., and Thomas, H. Pandore: a system to manage data distribution. In *Proc. of the 1990 International Conference on Supercomputing,* ACM, Jun. 1990.

[3] Bagheri, B., Raghavachari, B., Scott, L.R. PC-a parallel extension of C. Technical Report, Computer Science Department, Pennsylvania State University, Nov. 1990.

[4] Beguelin, A., Dongarra, J., Geist, A., Manchek, B., and Sunderam, V. A user's guide to PVM, parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Apr. 1991.

[5] Callahan, D. and Kennedy, K. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing* 2, 1988, 151-169.

[6] Carriero, N. and Gelernter, D. Linda in context. *Communications of the ACM,* 32(4), Apr. 1989, 444-458.

[7] Chen, M., Choo, Y., and Li, J. Crystal: from functional description to efficient parallel code. *Proc. from The Third Conference on Hypercube Concurrent Computers and Applications,* 1988, 417-33.

[8] Derby, T., Eskow, E., Neves, R., Rosing, M., Schnabel, R., and Weaver, R. The DINO User's Manual. Department of Computer Science Technical Report CU-CS-501-90, University of Colorado at Boulder, Nov. 1990.

[9] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. Fortran D language specification. Center for Research on Parallel Computation Technical Report CRPC-TR90079, Dec. 1990.

[10] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. *Solving Problems on Concurrent Computers,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[11] Geist, G.A., Heath, M.T., Peyton, B.W., and Worley, P.H. PICL: a portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, Jul. 1990.

[12] Gerndt, M. Updating distributed variables in local computations. *Concurrency: Practice & Experience,* 1990.

[13] Griswold, W.G., Harrison, G.A., Notkin, D., and Snyder, L. Scalable abstractions for parallel programming. *Proc. of the Fifth Distributed Memory Computing Conference,* IEEE Press, 1990, 1008-1016.

[14] Hatcher, P., Quinn, M., Lapadula, A., Seevers, B., Anderson, R., Jones, R. Data-parallel programming on MIMD computers. To appear in *IEEE Trans. on Parallel and Distributed Systems,* Jul. 1991.

[15] Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C. An overview of the Fortran D programming system. Center for Research on Parallel Computation Technical Report CRPC-TR91121, Mar. 1991.

[16] Intel iPSC Programmer's Reference Guide, Number 310612-002, Mar. 1987.

[17] Koelbel, C., Mehrotra, P., Saltz, J., and Berryman, S. Parallel loops on distributed machines. In *Proc. of the 5th Distributed Memory Computing Conference,* IEEE Press, 1990.

[18] Koelbel, C., Mehrotra, P., and Van Rosendale, J. Supporting shared data structures on distributed memory architectures. *Proc. of 2nd SIGPLAN Symposium on Principles and Practice of Parallel Processing.* ACM, 1990, 177-186.

[19] Li, G. and Coleman, T. A parallel triangular solver for a hypercube multiprocessor. *Hypercube Multiprocessors 1987,* SIAM, 1987, 539-551.

[20] Li, J. and Chen, M. Generating explicit communication from shared-memory program references. In *Proc. of Supercomputing '90,* 1990.

[21] Littlefield, R. Efficient iteration in data-parallel programs with irregular and dynamically distributed data structures. Technical Report 90-02-06, Department of Computer Science and Engineering, University of Washington, Feb. 1990.

[22] McGraw, J., Allan, S., Glauert, J., and Dobes, I. SISAL: streams and iteration in a single assignment language. Language Reference Manual. Technical Report M-146. Lawrence Livermore National Laboratory, 1983.

[23] Mehrotra, P. and Van Rosendale, J. Programming distributed memory architectures using Kali. ICASE Report 90-69, Institute for Computer Application in Science and Engineering, Hampton, VA, Oct. 1990.

[24] Mirchandaney, S., Saltz, J., Mehrotra, P., and Berryman, H. A scheme for supporting automatic data migration on multicomputers. In *Proc. of the 5th Distributed Memory Computing Conference,* IEEE Press, 1990.

[25] Quinn, M. and Hatcher, P. Data-parallel programming on multicomputers. *IEEE Software,* Sep. 1990, 69-76.

[26] Reeves, A. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NY, Jun. 1990.

[27] Rogers, A. and Pingali, K. Process decomposition through locality of reference. In *Proc. of the ACM SIGPLAN '89 Conference on Program Language Design and Implementation,* Jun. 1989.

[28] Rose, J., and Steele, G. C*: an extended C language for data parallel programming. Technical Report Pl85-7, Thinking Machines Corp., 1987.

[29] Rosing, M. Efficient language features for complex data parallelism on distributed memory multiprocessors. Ph.D. Thesis, Department of Computer Science, University of Colorado at Boulder, Aug. 1991.

[30] Rosing, M., Schnabel, R., and Weaver, R. The DINO parallel programming language. To appear in *The Journal of Parallel and Distributed Computing,* 1991.

[31] Rosing, M., Schnabel, R., and Weaver, R. Massive parallelism and process contraction in DINO. In Dongarra, J., Messina, P., Sorensen, D.C., and Voigt, R.G. (Eds.), *Parallel Processing for Scientific Computation,* SIAM, 1990, 364-369.

[32] Rosing, M., and Weaver, R. Mapping data to processors in distributed memory computers. *Proc. of Fifth Distributed Memory Computing Conference,* IEEE Press, 1990, 884-893.

[33] Ruppelt, T., and Wirtz, G. From mathematical specifications to parallel programs on a message based system. *Proc. 1988 International Conference on Supercomputing,* ACM, 1988, 108-118.

[34] Saltz, J., Berryman, H., and Wu, J. Multiprocessors and runtime compilation. ICASE Report 90-59, Institute for Computer Application in Science and Engineering, Hampton, VA, Sep. 1990.

[35] Socha, D.G. Spot: a data parallel language for iterative algorithms. Department of Computer Science Technical Report TR 90-03-01, University of Washington, 1990.

[36] Tseng, P.S. A parallelizing compiler for distributed memory parallel computers. In *Proc. of the ACM SIGPLAN '90 Conference on Program Language Design and Implementation,* Jun. 1990.

[37] Wu, J., Saltz, J., Berryman, H., and Hiranandani, S. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, Jan. 1991.

[38] Zima, H., Bast, H.-J., and Gerndt, M. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing, 6, 1986, 1-18.*